CSE 341: Homework 3

due 02/05/01

All functions you are asked to implement and turn in should be typed and tested with Miranda. With each function turn in the source code listing as well as sample output for a demonstrative set of tests. Do not forget to test *end cases*. Use the `turnin` program to turn in an electronic copy of your source code.

Part I: (Not to be turned in.)

Try these before section.

Suppose that the following Miranda script has been filed in:

```
cube x = x*x*x
my_map f [] = []
my_map f (x:xs) = f x : my_map f xs
my_append [] x = x
my_append (x:xs) ys = x : my_append xs ys
my_map2 f [] [] = []
my_map2 f (a:as) (b:bs) = f a b : my_map2 f as bs
rev f x y = f y x
alligator x y = 3+y
```

What is the result of evaluating the following Miranda expressions? (If there is a compile-time type error, or a run-time error, or a non-terminating computation, say so.) If the expression is followed by :: then give the type, instead of the value.

```
cube 3
cube ::
my_map ::
my_map cube ::
my_map cube [1..]
my_append ::
my_map2 alligator [1..]  [10..]
my_map2 ::
my_map2 alligator ::
my_map2 (rev alligator) ::
alligator (1/0) (6/2)
alligator (6/2) (1/0)
alligator ::
(rev alligator) 10 20
rev ::
rev my_map2::
```

Part II: (Not to be turned in.)

These are some sample problems that we will be discussing in section.

1. Write a function `midpoint` that takes in two tuples containing two numbers each and computes and returns the midpoint. For example:

```
midpoint (2,0) (8,2)                 => (5.0,1.0)
midpoint (-5,1) (5,-1)               => (0.0,0.0)
```

2. Write a Miranda function `no_duplicates` that takes a list of numbers as an argument, and returns a new list which is like the argument but with duplicate elements dropped. The order of the elements in the returned list doesn't matter. For example:

```
no_duplicates [1, 3, 3, 8, 5, 3, 5]  => [1, 3, 8, 5]
no_duplicates [1, 3, 5]  => [1, 3, 5]
no_duplicates []  => []
```

3. (5 points) Define an infinite data structure that contains *Pascal's triangle*. For example:

```
take 5 pascal           => [[1],[1,1],[1,2,1],[1,3,3,1],[1,4,6,4,1]]
```

Here, `pascal` is an infinite data structure. The `take 5` bit extracts the first 5 elements from the infinite list and only prints them. Otherwise, if we tried to print `pascal` we would run out of heap space. Hint: define a helper function `next_pascal` that takes in one list corresponding to a row in Pascal's triangle and creates the next row. Pascal's triangle can be visualized as follows:

```
                  1
                1   1
              1   2   1
            1   3   3   1
          1   4   6   4   1
        1   5  10  10   5   1
      1   6  15  20  15   6   1
    1   7  21  35  35  21   7   1
  1   8  28  56  70  56  28   8   1
1   9  36  84 126 126  84  36   9   1
```

Notice that the 56 to the right of the 70 on the second to bottom row can be calculated by adding the 21 and the 35 that are above it on the previous row. This gives us the recursive definition for how to compute a row of this triangle given the preceding row.

Part III: (Due Monday, 02/05/01. Turn this in.)

Complete the following. You do not have to handle errors in input gracefully. **As usual, you must turn in a hardcopy of your code and output of your code on a demonstrative set of test cases. Also use the `turnin` program to submit an electronic copy of your code.**

1. (5 points) Write a function `repeat_list` that takes a number, k, and a list and returns a new list with each element repeated k times. For example:

```
repeat_list 3 [1..3]
      => [1,1,1,2,2,2,3,3,3]
```

2. (5 points) Write a function `insert_list` that takes two lists (possibly lists of characters, I.e. strings) and returns a list of lists that is the first list inserted into the second list at all possible locations. For example:

```
insert_list "XX" "abcd"
      => ["XXabcd","aXXbcd","abXXcd","abcXXd","abcdXX"]
insert_list [0] [1..3]
      => [[0,1,2,3],[1,0,2,3],[1,2,0,3],[1,2,3,0]]
```

3. (5 points) Create the infinite list of balanced parentheses, `balanced_parens`. This list should contain, in increasing order with length, all strings containing only the characters `'('` and `')'` such that the parentheses are balanced. For example, `"(())()"` is balanced while `"()))())"` is not.

```
take 7 balanced_parens
      => ["","()","()()","(())","()()","()()()","(())()"]
member balanced_parens "()((()()))"
      => True
```

   It is ok for your list to contain the same string more than once; however, it must contain every string with balanced parenthesis. Note that there are an exponential number of strings with balanced parentheses of a given length. As such, expect your program to run out of heap space if you search, with `member`, for long parenthesized strings.

   Hint: Use `insert_list`.